一种基于动态块链结构的目录树存储方案

解决什么技术问题？为什么要做这件事？产品需求是什么？

本专利旨在解决分布式文件系统中元数据单机存储存储效率低下的技术问题，尤其是在高点查比例的工作负载下，现有基于 LSM-Tree 的 KV 存储引擎在元数据存储方面存在明显的性能瓶颈和资源浪费。

在你的创新之前业界的解决方案以及问题是什么？

业界在分布式文件系统元数据存储管理方面，最常见的解决方案是采用基于 LSM-Tree（如 LevelDB、RocksDB）的 KV 存储引擎。 LSM-Tree 因其写优化特性，能够支持高效的写入操作，在许多场景下表现良好。 然而，针对文件系统元数据管理这种特定场景，LSM-Tree 固有的架构缺陷开始显现，主要问题包括：

- **问题一：排序和合并（Compaction）开销大：**LSM-Tree 为了维护数据有序性，采用了基于 Sorted String Table (SSTable) 的多层结构。 新写入的数据首先进入内存中的 MemTable，当 MemTable 达到一定阈值后，会被刷写 (flush) 到磁盘，形成 level 0 的 SSTable。 后台 Compaction 进程会将 level 0 的 SSTable 与更高 levels 的 SSTable 进行合并和排序，生成新的 SSTable，并将旧的 SSTable 逐渐淘汰。
- **问题二：查询效率低：** LSM-Tree 在查找某个 Key 时，需要从内存中的 MemTable 开始，逐层遍历磁盘上的 SSTable。 由于 SSTable 分布在多个层级，并且可能存在重叠，查询过程需要在多个文件中进行查找，才能确定最终结果。
- **问题三：排序操作冗余：** LSM-Tree 强制维护全局有序的索引结构，所有写入的数据都需要按照 Key 的顺序进行组织和存储，Compaction 过程也需要对数据进行排序。而文件系统的 readdir 操作，仅仅需要返回一个目录下文件的列表，并不要求这个列表全局有序。这样LSM-Tree 强制执行的全局排序操作在文件系统元数据场景下，就成为了一种不必要的开销。

总结来说，现有基于 LSM-Tree 的 KV 存储引擎，虽然在通用 KV 存储场景下表现良好，但由于其**固有的排序合并机制和多层查找结构，与文件系统元数据“点查为主，局部范围查，无需全局排序”的访问特点存在错配**，导致性能和资源效率方面都存在显著的提升空间。

解决这个技术问题你的技术方案是什么？做这个产品你的创意是怎么做的？要体现出你的创意与现有技术方案或现有产品的区别，这个区别就是你的创意的创新点，也是重要保护点。

（这部分最重要，要详细写明DEMO，而不仅仅停留在一个idea，最好有图，用文字解释图）

我们提出了一种基于动态块链结构目录树存储引擎方案，专门为文件系统元数据管理场景设计。该方案**摒弃了 LSM-Tree 的全局有序和 Compaction 机制**，实现了高性能的元数据存储管理。

创新点提炼：

我们的方案与现有基于 LSM-Tree 的方案相比，主要创新点体现在以下几个方面：

- **创新点一：消除冗余排序，降低 Compaction 开销。**
- **创新点二：内存索引提升点查性能。**
- **创新点三：优化的 readdir 流程，提升 readdir 性能的同时 不牺牲一致性。**

整体技术方案：

1. 整体架构

- **元数据管理层**
  - 一级哈希表：目录ID → 二级Hash表地址及块链头节点地址 。
  - 二级哈希表：文件名 → 物理位置。
  - 空闲块管理器：按块类型维护位图索引（8K/16K/32K/64K）。
- **数据存储层**
  - 块池文件：按类型预分配固定容量文件 (如block_8k.bin) 。
  - 块链结构：目录元数据按插入顺序组织成动态块链 (可混合块类型) 。

2. 核心数据结构设计

2.1. 内存数据结构

- **块类型**

```c
// 块类型枚举（8KB/16KB/32KB/64KB ...）
enum  BlockType {
    BLOCK_8K   = 0,  // 8KB 块
    BLOCK_16K  = 1,  // 16KB 块
    BLOCK_32K  = 2,  // 32KB 块
    BLOCK_64K  = 3   // 64KB 块
```

```
    ...
};
```

- **一级哈希表**（目录ID → DirEntry）：

```c
// 目录元数据条目（一级哈希表 Value 类型）
struct DirEntry {
    uint64_t dir_id;              // 目录ID
    BlockType current_block_type; // 当前使用的块类型
    ChunkNode* chunk_chain;       // 块链头节点
    std::unordered_map<std::string, FileLocation>* file_map; // 文件名 → 位置
```

- **二级哈希表** （文件 / 目录名 → FileLocation）

```c
// 文件位置元数据（二级哈希表 Value 类型）
struct FileLocation {
    BlockType block_type; // 所属块类型
    uint32_t chunk_id;    // 块文件ID
    uint32_t slot;         // 块内槽位偏移（按记录大小对齐）
};
```

- **块链节点**（链表结构）：
  - 每个目录对应一个块链，按插入顺序组织。
  - 块链中可能存在混合块类型（如 8KB -> 16KB -> 64KB -> ...）。

```c
struct ChunkNode {
    uint32_t chunk_id;             // 物理块号
    BlockType current_type;        // 当前块类型（8K/16K/32K/64K）
    uint32_t remaining_offset;     // 块内剩余偏移量
    ChunkNode* next;               // 下一块指针
};
```

2.2 块池文件

- **按块类型分文件存储**：

```
/block_pool/
├── block_4k.bin    # 4KB块文件
├── block_8k.bin    # 8KB块文件
├── block_16k.bin   # 16KB块文件
├── block_32k.bin   # 32KB块文件
└── block_64k.bin   # 64KB块文件
...
```

- **块类型与容量：**

每条记录定长256B，所以一个4KB块，可以存放16条记录，其他类型块的相关数据估算如下：

块类型	记录容量（256B/条）	1万条所需块数	1百万条所需块数
8KB	32	313	31300
16KB	64	157	15700
32KB	128	79	7900
64KB	256	40	4000
128KB	512	20	2000
256KB	1024	10	1000
512KB	2048	5	500
1024KB	4096	3	300

3.系统执行流程

3.1. 写入流程

1. **索引查询：**通过目录ID找到一级Hash表中对应的二级Hash表以及元数据块链。
2. **空间检查**：检查当前尾块剩余空间。
3. **块大小升级**：若空间不足，分配更大块类型追加到链尾。
4. **数据写入**：将记录写入当前尾块的空闲槽位。
5. **索引更新**：在二级哈希表记录新文件的位置（文件ID + 块号 + 槽位）。

3.2. 删除流程

1. **索引查询：**
   a. 通过目录ID找到一级Hash表中对应的二级Hash表以及元数据块链。
   b. 通过文件名在二级哈希表找到记录对应的块类型、所在的块号及槽位。
2. **覆盖空洞**：

- 找到块链最后一个块的最后一个有效记录。
- 将其复制到待删除位置，更新最后一个块的remaining_offset，即remaining_offset -= 256B。

3. **索引维护**：
   - 删除原文件名索引
   - 更新被移动记录的索引
4. **释放空块**：
   - 若最后一个块全空，从块链中移除并释放回空闲池。

3.3. 点读流程

1. **索引查询**：
   - 通过目录ID找到一级Hash表中对应的二级Hash表
   - 通过文件名在二级哈希表查文件的物理位置。
2. **直接读取**：根据块类型+块号+槽位定位并读取数据。
3. **结果返回**：返回256B记录内容。

3.4. readdir 流程

3.4.1. Readdir需求

1. **POSIX 兼容性**：
   - 不保证readdir 调用的全局一致性（允许遍历过程中文件被增删改）。
   - 返回结果不能重复或者遗漏。
2. **分页遍历机制**：
   - 大目录无法一次性返回全部文件列表，需设计 游标（Cursor）机制 支持分页读取：
     - 游标记录遍历进度（如当前块位置、已返回记录标识等）。
     - 每次调用 readdir 返回一批记录（如每页 1000 条），并更新游标状态。

3.4.2. 即时碎片回收对 readdir 一致性的影响

删除操作会将尾部记录移动到前面的空洞，可能导致 readdir 读取时的遗漏或重复，具体来说：

1. **从前往后扫**：由于删除操作可能将尾部记录移动到前面的空洞，如果在扫描过程中发生这种移动，后续的记录可能已经被移动，导致当前块的后续记录被遗漏。例如，扫描到块A时，块A中的某个记录被删除，尾部记录被移动到块A的空洞处，而块A之后的块可能已经被处理，导致移动后的记录未被扫描到。

正向扫描 Plain Text

```
[块A: 空洞] → [块B: Y] → [块C: Z]
            ↓ 删除X并移动Z到块A ↓
[块A: Z'] → [块B: Y] → [块C: 空洞]
扫描顺序：A → B → C
结果：Z'未被扫描 → 遗漏Z。
```

2. **从后往前扫**：同样，删除操作可能导致记录移动，但方向相反。例如，扫描到块C时，块C中的记录被移动到前面的块A，导致块C中的记录已经被处理，而移动后的记录在块A中被再次扫描，导致重复。

逆向扫描 Plain Text

```
[块C: Z] → [块B: Y] → [块A: 空洞]
            ↓ 移动Z到块A ↓
[块C: 空洞] → [块B: Y] → [块A: Z']
扫描顺序: C → B → A
结果: 返回Z（旧位置）和Z'（新位置）→ 重复。
```

3.4.3. 方案核心思想

通过 **逆向扫描（从后往前）** 优先处理最新数据，结合去重机制，确保遍历过程中同一逻辑记录仅返回一次，同时避免遗漏。

关键设计：

- **逆向扫描**：从尾部块（最新块）开始扫描，降低补空洞操作对已扫描块的影响。
- **去重机制：**
  - 每个记录的尾部8字节存放最新修改时间戳。
  - 遍历开始时记录一个全局时间戳（global_ts），所有记录的可见性基于此global_ts判断，用于避免返回重复记录。

3.4.4. 方案实现步骤

3.4.4.1. 时间戳初始化

- **record_ts 写入：**
  - 每个记录的尾部固定包含 8 字节的 record_ts，表示其写入时间戳。
  - **写入规则：**
    - 新增记录时，record_ts 设为当前全局时间戳。
    - 移动记录（补空洞）时，record_ts 必须更新为当前时间戳（确保旧位置记录的 record_ts 失效）。
  - **global_ts 生成：**在首次 readdir 请求时，获取当前全局时间戳（单调递增），作为本次遍历的 global_ts。

3.4.4.2. 逆向扫描流程

1. **游标初始化：**从尾部块（最新块）开始扫描。
2. **记录过滤规则：**仅返回 record_ts ≤ global_ts 的记录 （去重）。
3. **游标推进：**扫描完当前块后，移动到前一个块（更旧的块），直到所有块处理完成。

3.4.4.3. 重复场景验证

```
初始状态：
块C（最新）：[Z, record_ts=100]
块B:          [Y, record_ts=90]
块A（最旧）：[空洞]

操作：
1. 删除块C中的记录Z（逻辑删除，标记为空洞）。
2. 将Z移动到块A的空洞处，更新其 `record_ts=200`。
新状态：
块C：[空洞]
块B：[Y, record_ts=90]
块A：[Z', record_ts=200]

逆向扫描流程（global_ts=150）：
1. 扫描块C：无有效记录（Z已删除）。
2. 扫描块B：返回Y（record_ts=90 ≤ 150）。
3. 扫描块A：Z'的record_ts=200 > 150 → **过滤不返回**。
结果：仅返回Y，无重复。
```

4. 你的技术方案或创新产品有什么优点？

该方案具有以下优点：

- **极致的点查性能：** 两级哈希结构保证点查操作只需2次内存操作和一次 IO，大幅降低点查延迟。
- **高效的范围查询：** readdir 操作只需扫描块链，同时基于动态块链技术减少了块链的长度，无需像 LSM-Tree 那样遍历多层 SSTable，提升了范围查询效率。
- **极低的 Compaction 开销：** 删除操作通过移动块链尾部记录覆盖空洞的方式回收空间，消除额外 Compaction 开销。